infrared Documentation

Release 2.0.1.dev691

Yair Fried

Jun 26, 2017

Core

InfraRed is a plugin based system that aims to provide an easy-to-use CLI for Ansible based projects. It aims to leverage the power of Ansible in managing / deploying systems, while providing an alternative, fully customized, CLI experience that can be used by anyone, without prior Ansible knowledge.

The project originated from Red Hat OpenStack infrastructure team that looked for a solution to provide an "easier" method for installing OpenStack from CLI but has since grown and can be used for *any* Ansible based projects.

CHAPTER 1

Welcome to infrared's documentation!

Bootstrap

Setup

Clone *infrared* 2.0 from GitHub:

```
git clone https://github.com/redhat-openstack/infrared.git
```

Setup virtualenv and install from source using pip:

```
cd infrared
virtualenv .venv && source .venv/bin/activate
pip install --upgrade pip
pip install --upgrade setuptools
pip install .
```

Warning: It's important to upgrade `pip` first, as default `pip` version in RHEL (1.4) might fail on dependencies

Note: *infrared* will create a default workspace for you. This workspace will manage your environment details.

Provision

In this example we'll use virsh provisioner in order to demonstrate how easy and fast it is to provision machines using *infrared*.

Add the virsh plugin:

```
infrared plugin add plugins/virsh
```

Print *virsh* help message and all input options:

```
infrared virsh --help
```

For basic execution, the user should only provide data for the mandatory parameters, this can be done in two ways:

1. *CLI*
2. *Answers File*

CLI

Notice the only three mandatory paramters in *virsh* provisioner are:

- `--host-address` - the host IP or FQDN to ssh to
- `--host-key` - the private key file used to authenticate to your `host-address` server
- `--topology-nodes` - type and role of nodes you would like to deploy (e.g: `controller:3` == 3 VMs that will act as controllers)

We can now execute the provisioning process by providing those parameters through the CLI:

```
infrared virsh --host-address $HOST --host-key $HOST_KEY --topology-nodes
↪"undercloud:1,controller:1,compute:1"
```

That is it, the machines are now provisioned and accessible:

```
TASK [update inventory file symlink] *******************************************
                    [[ previous task time: 0:00:00.306717 = 0.31s / 209.71s ]]
changed: [localhost]

PLAY RECAP *********************************************************************
compute-0                  : ok=4    changed=3    unreachable=0    failed=0
controller-0               : ok=5    changed=4    unreachable=0    failed=0
localhost                  : ok=4    changed=3    unreachable=0    failed=0
undercloud-0               : ok=4    changed=3    unreachable=0    failed=0
hypervisor                   : ok=85   changed=29   unreachable=0    failed=0

                    [[ previous task time: 0:00:00.237104 = 0.24s / 209.94s ]]
                    [[ previous play time: 0:00:00.555806 = 0.56s / 209.94s ]]
               [[ previous playbook time: 0:03:29.943926 = 209.94s / 209.94s ]]
                    [[ previous total time: 0:03:29.944113 = 209.94s / 0.00s ]]
```

Note: You can also use the auto-generated ssh config file to easily access the machines

Answers File

Unlike with *CLI*, here a new answers file (INI based) will be created. This file contains all the default & mandatory parameters in a section of its own (named `virsh` in our case), so the user can easily replace all mandatory parameters. When the file is ready, it should be provided as an input for the `--from-file` option.

Generate Answers file for *virsh* provisioner:

```
infrared virsh --generate-answers-file virsh_prov.ini
```

Review the config file and edit as required:

Listing 1.1: virsh_prov.ini

```
[virsh]
host-key = Required argument. Edit with any value, OR override with CLI: --host-key=
↪<option>
host-address = Required argument. Edit with any value, OR override with CLI: --host-
↪address=<option>
topology-nodes = Required argument. Edit with one of the allowed values OR override␣
↪with CLI: --topology-nodes=<option>
host-user = root
```

Note: `host-key`, `host-address` and `topology-nodes` don't have default values. All arguments can be edited in file or overridden directly from CLI.

Note: Do not use double quotes or apostrophes for the string values in the answers file. *Infrared* will NOT remove those quotation marks that surround the values.

Edit mandatory parameters values in the answers file:

```
[virsh]
host-key = ~/.ssh/id_rsa
host-address = my.host.address
topology-nodes = undercloud:1,controller:1,compute:1
host-user = root
```

Execute provisioning using the newly created answers file:

```
infrared virsh --from-file=virsh_prov.ini
```

Note: You can always overwrite parameters from answers file with parameters from CLI:

```
.. code-block:: text
```

infrared virsh –from-file=virsh_prov.ini –topology-nodes="undercloud:1,controller:1,compute:1,ceph:1"

Done. Quick & Easy!

Installing

Now let's demonstrate the installation process by deploy an OpenStack environment using RHEL-OSP on the nodes we have provisioned in the previous stage.

Undercloud

Just like in the provisioning stage, here also the user should take care of the mandatory parameters (by CLI or INI file) in order to be able to start the installation process. Let's deploy a TripleO Undercloud:

```
infrared tripleo-undercloud --version 10 --images-task rpm
```

This will deploy OSP 10 (`Newton`) on the node `undercloud-0` provisioned previously.

Overcloud

Let's deploy a TripleO Overcloud:

```
infrared tripleo-overcloud --deployment-files virt --version 10 --introspect yes --
↪tagging yes --deploy yes --post yes
```

This will deploy OSP 10 (`Newton`) overcloud from the undercloud defined previously previously. Given the topology defined by the *Answers File* earlier, the overcloud should contain: - 1 controller - 1 compute - 1 ceph storage

Setup

Supported distros

Currently supported distros are:

- Fedora 22, 23, 24, 25
- RHEL 7.2 (best effort, deprecated)
- RHEL 7.3

Warning: Python 2.7 and virtualenv are required.

Prerequisites

Warning: sudo or root access is needed to install prerequisites!

General requirements:

```
sudo yum install git gcc libffi-devel openssl-devel
```

Note: Dependencies explained:

- git - version control of this project
- gcc - used for compilation of C backends for various libraries
- libffi-devel - required by cffi
- openssl-devel - required by cryptography

Closed Virtualenv is required to create clean python environment separated from system:

```
sudo yum install python-virtualenv
```

Ansible requires python binding for SELinux:

```
sudo yum install libselinux-python
```

otherwise it won't be able to run modules with copy/file/template functions!

Note: libselinux-python is in *Prerequisites* but doesn't have a pip package. It must be installed on system level.

Note: Ansible requires also **libselinux-python** installed on all nodes using copy/file/template functions. Without this step all such tasks will fail!

Virtualenv

`infrared` shares dependencies with other OpenStack products and projects. Therefore there's a high probability of conflicts with python dependencies, which would result either with `infrared` failure, or worse, with breaking dependencies for other OpenStack products. When working from source, virtualenv usage is recommended for avoiding corrupting of system packages:

```
virtualenv .venv
source .venv/bin/activate
pip install --upgrade pip
pip install --upgrade setuptools
```

Warning: Use of latest ''pip'' is mandatory, especially on RHEL platform!

Note: On Fedora 23 with EPEL repository enabled, RHBZ#1103566 also requires

```
dnf install redhat-rpm-config
```

Installation

Clone stable branch from Github repository:

```
git clone https://github.com/redhat-openstack/infrared.git
```

Install `infrared` from source:

```
cd infrared
pip install .
```

Note: For development work it's better to install in editable mode and work with master branch:

```
pip install -e .
```

Configuration

infrared allows to change default options such as:

- `workspaces_base_folder`: the workspaces base folder
- `plugins_conf_file`: the plugins configuration file path

This can be done by creating a configuration file, `./infrared.cfg`, and overriding default values in it:

```
cp infrared.cfg.example infrared.cfg
```

When `infrared.cfg` file is not found, default options defined in `infrared.cfg.example` file will be used:

```
[core]
# the base folder where all the workspaces will be stored
workspaces_base_folder = .workspaces

# the plugins configuration file
plugins_conf_file = .plugins.ini
```

Ansible configuration and limitations

Usually *infrared* does not touch the settings specified in the ansible configuration file (`ansible.cfg`), with few exceptions.

Internally *infrared* use Ansible environment variables to set the directories for common resources (callback plugins, filter plugins, roles, etc); this means that the following keys from the Ansible configuration files are ignored:

- `callback_plugins`
- `filter_plugins`
- `roles_path`

It is possible to define custom paths for those items setting the corresponding environment variables:

- `ANSIBLE_CALLBACK_PLUGINS`
- `ANSIBLE_FILTER_PLUGINS`
- `ANSIBLE_ROLES_PATH`

Workspaces

With *workspaces*, user can manage multiple environments created by *infrared* and alternate between them. All runtime files (Inventory, hosts, ssh configuration, ansible.cfg, etc...) will be loaded from a workspace directory and all output files (Inventory, ssh keys, environment settings, facts caches, etc...) will be generated into that directory.

Create: Create new workspace. If name isn't provided, *infrared* will generate one based on timestamp:

```
infrared workspace create example

Workspace 'example' added
```

Inventory: Fetch workspace inventory file (a symlink to the real file that might be changed by *infrared* executions):

```
infrared workspace inventory

/home/USER/.infrared/workspaces/example/hosts
```

Checkout Creates new workspace if needed and switches to it:

```
infrared workspace checkout example3

Workspace 'example3' added
Now using workspace: 'example3'
```

Note: Checked out workspace is tracked via a status file in workspaces_dir, which means checked out workspace is persistent across shell sessions. You can pass checked out workspace by environment variable `IR_WORKSPACE`, which is non persistent

```
ir workspace list
| Name    | Is Active    |
|---------+--------------|
| bee     | True         |
| zoo     |              |

IR_WORKSPACE=zoo ir workspace list
| Name    | Is Active    |
|---------+--------------|
| bee     |              |
| zoo     | True         |

ir workspace list
| Name    | Is Active    |
|---------+--------------|
| bee     | True         |
| zoo     |              |
```

Warning: While `IR_WORKSPACE` is set *ir workspace checkout* is disabled

```
export IR_WORKSPACE=zoo
ir workspace checkout zoo
ERROR   'workspace checkout' command is disabled while IR_WORKSPACE␣
↪environment variable is set.
```

List: List all workspaces. Active workspace will be marked.:

```
infrared workspace list

| Name         | Is Active    |
|--------------+--------------|
| example      |              |
```

```
| example2    | True        |
| rdo_testing |             |
```

Delete: Deletes a workspace:

```
infrared workspace delete example

Workspace 'example' deleted
```

Cleanup: Removes all the files from workspace. Unlike delete, this will keep the workspace namespace and keep it active if it was active before.:

```
infrared workspace cleanup example2
```

Export: Package workspace in a tar ball that can be shipped to, and loaded by, other *infrared* instances:

```
infrared workspace export

Workspace example2 exported to example2.tar
```

To export non-active workspaces, or control the output file:

```
infrared workspace export example1 --dest /tmp/look/at/my/workspace

Workspace example1 exported to /tmp/look/at/my/workspace
```

Import: Load a previously exported workspace (local or remote):

```
infrared workspace import /tmp/look/at/my/new-workspace.tgz
infrared workspace import http://free.ir/workspaces/newworkspace.tgz

Workspace new-workspace was imported
```

Control the workspace name:

```
infrared workspace import /tmp/look/at/my/new-workspace --name example3

Workspace example3 was imported
```

Node list: List nodes, managed by a specific workspace:

```
infrared workspace node-list
| Name          | Address     |
|---------------+-------------|
| controller-0 | 172.16.0.94 |
| controller-1 | 172.16.0.97 |

infrared workspace node-list --name some_workspace_name
```

Note: To change the directory where Workspaces are managed, edit the `workspaces_base_folder` option. Check the Infrared Configuration for details.

Plugins

In *infrared* 2.0, *plugins* are self contained Ansible projects. They can still also depend on common items provided by the core project. Any ansible project can become an‘infrared‘ plugin by adhering to the following structure (see tests/example for an example plugin):

```
tests/example
- main.yml                # Main playbook. All execution starts here
- plugin.spec             # Plugin definition
- roles                   # Add here roles for the project to use
|   - example_role
|       - tasks
|           - main.yml
```

Note: This structure will work without any `ansible.cfg` file provided (unless common resources are used), as Ansible will search for references in the relative paths described above. To use an `ansible.cfg` config file, use absolute paths to the plugin directory.

Plugin structure

Main entry

infrared will look for a playbook called `main.yml` to start the execution from.

Plugins are regular Ansible projects, and as such, they might include or reference any item (files, roles, var files, ansible plugins, modules, templates, etc...) using relative paths to current playbook. They can also use roles, callback and filter plugins defined in the common/ directory provided by *infrared* core.

An example of `plugin_dir/main.yml`:

```
- name: Main Play
  hosts: all
  vars_files:
      - vars/some_var_file.yml
  roles:
    - role: example_role
  tasks:
      - name: fail if no vars dict
        when: "provision is not defined"
        fail:

      - name: fail if input calls for it
        when: "provision.foo.bar == 'fail'"
        fail:

      - debug:
          var: inventory_dir
        tags: only_this

      - name: Test output
        vars:
          output_file: output.example
        file:
          path: "{{ inventory_dir }}/{{ output_file }}"
```

```
            state: touch
        when: "{{ provision is defined }}"
```

Plugin Specification

infrared gets all plugin info from `plugin.spec` file. Following *YAML* format. This file define the CLI this plugin exposes, its name and its type.

```
plugin_type: provision
subparsers:
    example:
        description: Example provisioner plugin
        include_groups: ["Ansible options", "Inventory", "Common options", "Answers file"]
        groups:
            - title: Group A
              options:
                  foo-bar:
                      type: Value
                      help: "foo.bar option"
                      default: "default string"

                  dictionary-val:
                      type: KeyValueList
                      help: "dictionary-val option"

            - title: Group B
              options:
                  iniopt:
                      type: IniType
                      help: "Help for '--iniopt'"
                      action: append

            - title: Group C
              options:
                  uni-dep:
                      type: Value
                      help: "Help for --uni-dep"
                      required_when: "req-arg-a == yes"

                  multi-dep:
                      type: Value
                      help: "Help for --multi-dep"
                      required_when:
                          - "req-arg-a == yes"
                          - "req-arg-b == yes"

                  req-arg-a:
                      type: Bool
                      help: "Help for --req-arg-a"

                  req-arg-b:
                      type: Bool
                      help: "Help for --req-arg-b"
```

Plugin type can be one of the following: `provision`, `install`, `test`, `other`.

To access the options defined in the spec from your playbooks and roles use the plugin type with the option name. For example, to access `dictionary-val` use `{{ provision.dictionary.val }}`.

Note: the vars-dict defined by *Complex option types* is nested under `plugin_type` root key, and passed to Ansible using `--extra-vars` meaning that any vars file that has `plugin_type` as a root key, will be overriden by that vars-dict. See Ansible variable precidence for more details.

Include Groups

A plugin can reference preset control arguments to be included in its CLI

Answers File: Instead of explicitly listing all CLI options every time, *infrared* plugins can read their input from `INI` answers file, using `--from-file` switch. use `--generate-answers-file` switch to generate such file. It will list all input arguments a plugin accepts, with their help and defaults. CLI options still take precedence if explicitly listed, even when `--from-file` is used.

Common Options:

- `--dry-run`: Don't execute Ansible playbook. Only write generated vars dict to stdout
- `--output`: Redirect generated vars dict from stdout to an explicit file (YAML format).
- `--extra-vars`: Inject custom input into the vars dict

Inventory: Load a new inventory to active workspace. The file is copied to workspace directory so all `{{ inventory_dir }}` references in playbooks still point to workspace directory (and not to the input file's directory).

Note: This file permanently becomes the workspace's inventory. To revert to original workspace the workspace must be cleaned.

Ansible options:

- `--verbose`: Set ansible verbosity level
- `--ansible-args`: Pass all subsequent input to Ansible as raw arguments. This is for power-users wishing to access Ansible functionality not exposed by *infrared*:

```
infrared [...] --ansible-args step;tags=tag1,tag2;forks=500
```

Is the equivalent of:

```
ansible-playbook [...] --step --tags=tag1,tag2 --forks 500
```

Complex option types

Infrared extends argparse with the following option types. These options are nested into the vars dict that is later passed to Ansible as extra-vars.

- **Value:** String value.
- **Bool:** Boolean value. Accepts any form of YAML boolean: `yes`/`no`, `true`/`false` `on`/`off`. Will fail if the string can't be resolved to this type.

- **IniType:** Value is in `section.option=value` format. `append` is the default action for this type, so users can provide multiple args for the same parameter.
- **KeyValueList:** String representation of a flat dict `--options option1:value1,option2:value2` becomes:

```
{"options": {"option1": "value1",
             "option2": "value2"}}
```

The nesting is done in the following manner: option name is split by − delimiter and each part is a key of a dict nested in side the previous one, starting with "plugin_type". Then value is nested at the inner-most level. Example:

```
infrared example --foo-bar=value1 --foo-another-bar=value2 --also_foo=value3
```

```
{
    "provision": {
        "foo": {
            "bar": "value1",
            "another": {
                "bar": "value2"
            }
        },
        "also_foo": "value3"
    }
}
```

- **FileValue** The absolute or relative path to a file. Infrared validates whether file exists and transform the path to the absolute.
- **VarFile**

  Same as the `FileValue` type but additionally Infrared will check the following locations for a file:

  - `argument/name/option_value`
  - `<spec_root>/defaults/argument/name/option_value`
  - `<spec_root>/var/argument/name/option_value`

  In the example above the CLI option name is `--argument-name`. The VarFile suites very well to describe options which point to the file with variables.

  For example, user can describe network topologies parameters in separate files. In that case, all these files can be put to the `<spec_root>/defaults/network` folder, and plugin specification can look like:

```
plugin_type: provision
subparsers:
my_plugin:
    description: Provisioner virtual machines on a single Hypervisor using␣
↪libvirt
    groups:
        - title: topology
          options:
              network:
                  type: VarFile
                  help: |
                      Network configuration to be used
                      __LISTYAMLS__
                  default: defautl_3_nets
```

Then, the cli call can looks simply like:

```
infrared my_plugin --network=my_file
```

Here, the 'my_file' file should be present in the `/{defaults|var}/network` folder, otherwise an error will be displayed by the Infrared. Infrared will transform that option to the absolute path and will put it to the provision.network variable:

```
provision.network: /home/user/..../my_plugin/defaults/my_file
```

That variable is later can be used in Ansible playbooks to load the appropriate network parameters.

Note: Infrared automatically checks for files with .yml extension. So the `my_file` and `my_file.yml` will be validated.

- **ListOfVarFiles** The list of files. Same as `VarFile` but represents the list of files delimited by comma (`,`).
- **VarDir** The absolute or relative path to a directory. Same as `VarFile` but points to the directory instead of file

Placeholders

Placeholders allow users to add a level of sophistication in options help field.

- **`__LISTYAMLS__`:** Will be replaced with a list of available YAML (`.yml`) file from the option's settings dir. | Assume a plugin with the following directory tree is installed:

```
plugin_dir
- main.yml                  # Main playbook. All execution starts here
- plugin.spec                  # Plugin definition
- vars                      # Add here variable files
    - yamlsopt
    |   - file_A1.yml       # This file will be listed for yamlsopt
    |   - file_A2.yml       # This file will be listed also for yamlsopt
    - another
        -yamlsopt
            - file_B1.yml  # This file will be listed for another-yamlsopt
            - file_B2.yml  # This file will be listed also for another-
↪yamlsopt
```

Content of `plugin_dir/plugin.spec`:

```
plugin_type: provision
description: Example provisioner plugin
subparsers:
    example:
        groups:
            - title: GroupA
                  yamlsopt:
                     type: Value
                     help: |
                           help of yamlsopt option
                           __LISTYAMLS__

                  another-yamlsopt:
                     type: Value
```

```
                    help: |
                          help of another-yamlsopt option
                          __LISTYAMLS__
```

Execution of help command (`infrared example --help`) for the 'example' plugin, will produce the following help screen:

```
usage: infrared example [-h] [--another-yamlsopt ANOTHER-YAMLSOPT]
                            [--yamlsopt YAMLSOPT]

optional arguments:
  -h, --help            show this help message and exit

GroupA:
  --another-yamlsopt ANOTHER-YAMLSOPT
                        help of another-yamlsopt option
                        Available values: ['file_B1', 'file_B2']
  --yamlsopt YAMLSOPT   help of yamlsopt option
                        Available values: ['file_A1', 'file_A2']
```

Required Arguments

InfraRed provides the ability to mark an argument in a specification file as 'required' using two flags:

1. 'required' - A boolean value tell whether the arguments required or not. (default is 'False')
2. 'required_when' - Makes this argument required only when the mentioned argument is given and has the exact mentioned value. (More than one condition is allowed with YAML list style)

For example, take a look on the `plugin.spec` ('Group C') in *Plugin Specification*

Plugin Manager

The following commands are used to manage *infrared* plugins

Add: *infrared* will look for a plugin.spec file in the given source and register the plugin under the given plugin-type (when source is 'all', all available plugins will be installed):

```
infrared plugin add tests/example
infrared plugin add <git_url>
infrared plugin add all
```

List: List all available plugins, by type:

```
infrared plugin list

-------------
| Type      | Name    |
-------------
| provision | example |
-------------
| install   |         |
-------------
| test      |         |
-------------
```

```
infrared plugin list --available

---------------------------
| Type      | Name              | Installed |
---------------------------
| provision | example           |     *     |
|           | foreman           |           |
|           | openstack         |           |
|           | virsh             |           |
---------------------------
| install   | collect-logs      |           |
|           | packstack         |           |
|           | tripleo-overcloud |           |
|           | tripleo-undercloud|           |
---------------------------
| test      | rally             |           |
|           | tempest           |           |
---------------------------
```

Note: Supported plugin types are defined in plugin settings file which is auto generated. Check the Infrared Configuration for details.

Remove: Remove the given plugin (when name is 'all', all plugins will be removed):

```
infrared plugin remove example
infrared plugin remove all
```

Execute: Plugins are added as subparsers under `plugin type` and will execute the `main.yml` playbook:

```
infrared example
```

Topology

A topology is a description of an environment you wish to provision. We have divided it into two, *network topology* and *nodes topology*.

Nodes topology

Before creating our environment, we need to decide how many and what type of nodes to create. The following format is used to provide topology nodes:

```
infrared <provisioner_plugin> --topology-nodes NODENAME:AMOUNT
```

where `NODENAME` refers to files under `vars/topology/nodes/NODENAME.yml` (or `defaults/topology/nodes/NODENAME.yml`) and `AMOUNT` refers to the amount of nodes from the `NODENAME` we wish to create.

For example, if we choose the Virsh provisioner:

```
infrared virsh --topology-nodes undercloud:1,controller:3 ...
```

The above command will create 1 VM of type `undercloud` and 3 VMs of type `controller`

For any node that is provided in the CLI `--topology-nodes` flag, *infrared* looks for the node first under `vars/topology/nodes/NODENAME.yml` and if not found, under `default/topology/nodes/NODENAME.yml` where we supply a default set of supported / recommended topology files.

Lets examine the structure of topology file (located: var/topology/nodes/controller.yml):

```
name: controller        # the name of the VM to create, in case of several of the same␣
↪type, appended with "-#"
prefix: null            # in case we wish to add a prefix to the name
cpu: "4"                # number of vCPU to assign for the VM
memory: "8192"          # the amount of memory
swap: "0"               # swap allocation for the VM
disks:                  # number of disks to create per VM
    disk1:              # the below values are passed `as is` to virt-install
        import_url: null
        path: "/var/lib/libvirt/images"
        dev: "/dev/vda"
        size: "40G"
        cache: "unsafe"
        preallocation: "metadata"
interfaces:             # define the VM interfaces and to which network they should be␣
↪connected
    nic1:
        network: "data"
    nic2:
        network: "management"
    nic3:
        network: "external"
external_network: management  # define what will be the default external network
groups:                       # ansible groups to assign to the newly created VM
    - controller
    - openstack_nodes
    - overcloud_nodes
    - network
```

For more topology file examples, please check out the default available nodes

To override default values in the topology dict the extra vars can be provided through the CLI. For example, to add more memory to the controller node, the `override.controller.memory` value should be set:

```
infrared virsh --topology-nodes controller:1,compute:1 -e override.controller.
↪memeory=30720
```

Network topology

Before creating our environment, we need to decide number and types of networks to create. The following format is used to provide topology networks:

```
infrared <provisioner_plugin> --topology-network NET_TOPOLOGY
```

where `NET_TOPOLOGY` refers to files under `vars/topology/network/NET_TOPOLOGY.yml` (or if not found, `defaults/topology/network/NET_TOPOLOGY.yml`)

To make it easier for people, we have created a default network topology file called: `3_nets.yml` (you can find it under each provisioner plugin defaults/topology/network/3_nets.yml) that will be created automatically.

For example, if we choose the Virsh provisioner:

```
infrared virsh --topology-network 3_nets ...
```

The above command will create 3 networks: (based on the specification under `defaults/topology/network/3_nets.yml`)

data network - an isolated network # management network - NAT based network with a DHCP # external network - NAT based network with DHCP

If we look in the `3_nets.yml` file, we will see this:

```
networks:
    net1:
        <snip>
    net2:
        name: "management"                  # the network name
        external_connectivity: yes          # whether we want it externally accessible
        ip_address: "172.16.0.1"            # the IP address of the bridge
        netmask: "255.255.255.0"
        forward:                            # forward method
            type: "nat"
        dhcp:                               # omit this if you don't want a DHCP
            range:                          # the DHCP range to provide on that network
                start: "172.16.0.2"
                end: "172.16.0.100"
            subnet_cidr: "172.16.0.0/24"
            subnet_gateway: "172.16.0.1"
        floating_ip:                        # whether you want to "save" a range for␣
↪assigning IPs
            start: "172.16.0.101"
            end: "172.16.0.150"
    net3:
        <snip>
```

Interactive SSH

This plugin allows users to establish interactive ssh session to a host managed by *infrared*. To do this use:

```
infrared ssh <nodename>
```

where 'nodename' is a hostname from inventory file.

For example

```
infrared ssh controller-0
```

New In infrared 2.0

Highlights

1. **Workspaces:** Added Workspaces. Every session must be tied to an active workspace. All input and output file are taken from, and written to, the active workspace directory. which allows easy migration of workspace, and avoids accidental overwrites of data, or corrupting the working directory. The deprecates `ir-archive` in favor of `workspace import` and `workspace export`

2. **Stand-Alone Plugins:** Each plugins is fully contained within a single directory. Plugin structure is fully defined and plugins can be loaded from any location on the system. *"Example plugin"* shows contributors how to structure their Ansible projects to plug into *infrared*
3. **SSH:** Added ability to establish interactive ssh connection to nodes, managed by workspace using workspace's inventory `infrared ssh <hostname>`
4. **Single Entry-Point:** `ir-provisioner`, `ir-installer`, `ir-tester` commands are deprecated in favor of a single `infrared` entry point (`ir` also works). Type `infrared --help` to get the full usage manual.
5. **TripleO:**

   `ir-installer ospd` was broken into two new plugins:

   - TripleO Undercloud: Install undercloud up-to and including overcloud image creation
   - TripleO Overcloud: Install overcloud using an exsiting undercloud.
6. **Answers file:** The switch `--generate-conf-file` is renamed `--generate-answers-file` to avoid confusion with configuration files.
7. **Topoloy:** The topology input type has been deprecated. Use *KeyValueList* to define node types and amounts, and `include_vars` to add relevant files to playbooks, see Topology description for more information
8. **Cleanup:** the `--cleanup` options now accepts boolean values. Any YAML boolean is accpeted ("yes/no", "true/false", "on/off")
9. **Bootstrap:** On virtual environmants, tripleo-undercloud can create a snapshot out of the undercloud VM that can later be used to bypass the installation process.

Example Script Upgrade

infrared v2

```
## CLEANUP ##
infrared virsh -v -o cleanup.yml \
    --host-address example.redhat.com \
    --host-key ~/.ssh/id_rsa \
    --cleanup yes

## PROVISION ##
infrared virsh -v \
    --topology-nodes undercloud:1,
↪controller:1,compute:1 \
    --host-address example.redhat.com \
    --host-key ~/.ssh/id_rsa \
    --image-url http://www.images.com/
↪rhel-7.qcow2

## UNDERCLOUD ##
infrared tripleo-undercloud -v mirror␣
↪tlv \
    --version 9 \
    --build passed_phase1 \
    --ssl true \
    --images-task rpm

## OVERCLOUD ##
infrared tripleo-overcloud -v \
    --version 10 \
    --introspect yes \
    --tagging yes \
    --deploy yes \
    --post yes \
    --deployment-files virt \
    --network-backend vxlan \
    --overcloud-ssl false \
    --network-protocol ipv4

## TEMPEST ##
infrared tempest -v \
    --config-options "image.http_
↪image=http://www.images.com/cirros.
↪qcow2" \
    --openstack-installer tripleo \
    --openstack-version 9 \
    --tests sanity

# Fetch inventory from active workspace
WORKSPACE=$(ir workspace list | awk '/*/
↪{print $2}')
ansible -i .workspaces/$WORKSPACE/hosts␣
↪all -m ping
```

infrared v1

```
## CLEANUP ##
ir-provisioner -d virsh -v \
  --topology-nodes=undercloud:1,
↪controller:1,compute:1 \
  --host-address=example.redhat.com \
  --host-key=~/.ssh/id_rsa \
  --image-url=www.images.com/rhel-7.
↪qcow2 \
  --cleanup

## PROVISION ##
ir-provisioner -d virsh -v \
  --topology-nodes=undercloud:1,
↪controller:1,compute:1 \
  --host-address=example.redhat.com \
  --host-key=~/.ssh/id_rsa \
  --image-url=http://www.images.com/rhel-
↪7.qcow2

## OSPD ##
ir-installer --debug mirror tlv ospd -v -
↪o install.yml\
  --product-version=9 \
  --product-build=latest \
  --product-core-build=passed_phase1 \
  --undercloud-ssl=true \
  --images-task=rpm \
  --deployment-files=$PWD/settings/
↪installer/ospd/deployment/virt \
  --network-backend=vxlan \
  --overcloud-ssl=false \
  --network-protocol=ipv4

ansible-playbook -i hosts -e @install.
↪yml \
  playbooks/installer/ospd/post_install/
↪create_tempest_deployer_input_file.yml

## TEMPEST ##
ir-tester --debug tempest -v \
  --config-options="image.http_
↪image=http://www.images.com/cirros.
↪qcow2" \
  --tests=sanity.yml

ansible -i hosts all -m ping
```

Advance Features

Injection points

Different people have different use cases which we cannot anticipate in advance. To solve (partially) this need, we structured our playbooks in a way that breaks the logic into standalone plays. Furthermore, each logical play can be overriden by the user at the invocation level.

Lets look at an example to make this point more clear. Looking at our `virsh` main playbook, you will see:

```
- include: "{{ provision_cleanup | default('cleanup.yml') }}"
  when: provision.cleanup|default(False)
```

Notice that the `include:` first tried to evaluate the variable `provision_cleanup` and afterwards defaults to our own cleanup playbook.

This condition allows users to inject their own custom cleanup process while still reuse all of our other playbooks.

Override playbooks

In this example we'll use a custom playbook to override our cleanup play and replace it with the process described above. First, lets create an empty playbook called: `noop.yml`:

```
---
- name: Just another empty play
  hosts: localhost
  tasks:
    - name: say hello!
      debug:
          msg: "Hello!"
```

Next, when invoking *infrared*, we will pass the variable that points to our new empty playbook:

```
infrared virsh --host-address $HOST --host-key $HOST_KEY --topology-nodes $TOPOLOGY --
↪cleanup yes -e provision_cleanup=noop.yml
```

Now lets run see the results:

```
PLAY [Just another empty play] ***************************************************

TASK [setup] *********************************************************************
ok: [localhost]

TASK [say hello!] ****************************************************************
                        [[ previous task time: 0:00:00.459290 = 0.46s / 0.47s ]]
ok: [localhost] => {
    "msg": "Hello!"
}

msg: Hello!
```

If you have a place you would like to have an injection point and one is not provided, please contact us.

Contact Us

Team

Tal Kammer	tkammer@redhat.com
Yair Fried	yfried@redhat.com
Mailing List	rhos-infrared@redhat.com

GitHub

Issues are tracked via GitHub. For any concern, please create a new issue.

IRC

We are available on `#infrared` irc channel on `freenode`.

Contribute

Sending patches

Changes to project are accepted via review.gerrithub.io. Only members of `rhosqeauto-core` on group on GerritHub or `redhat-openstack` (RDO) organization on GitHub can submit patches. ask any of the current members about it.

You can use git-review (dnf/yum/pip install). To initialize the directory of `infrared` execute `git review -s`. Every patch needs to have *Change-Id* in commit message (`git review -s` installs post-commit hook to automatically add one).

For some more info about git review usage, read GerritHub Intro and OpenStack Infra Manual.

OVB deployment

Deploy TripleO OpenStack on virtual nodes provisioned from an OpenStack cloud

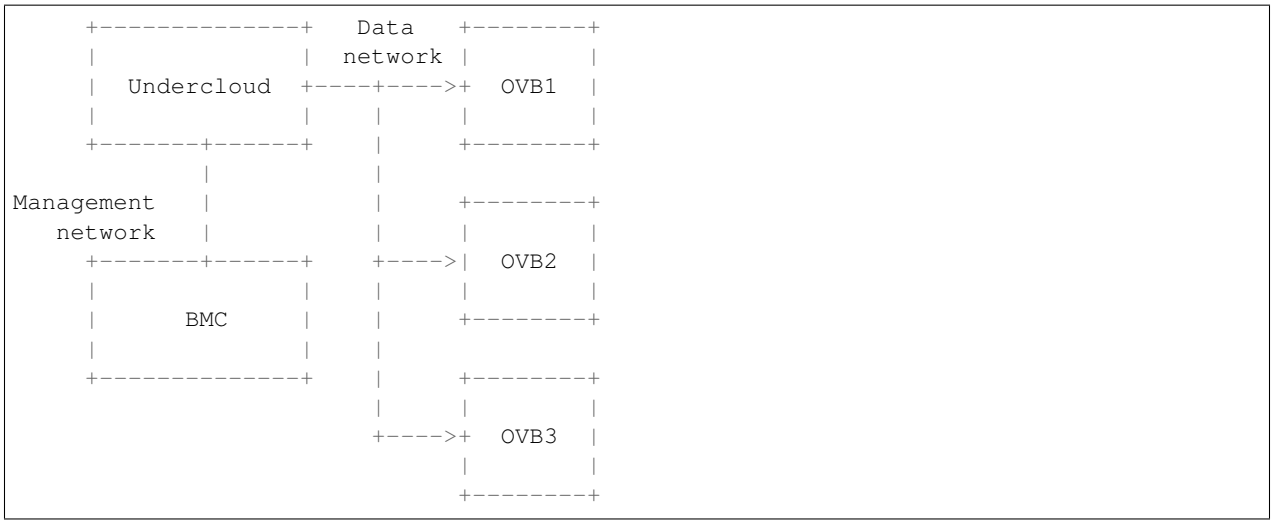
In a TripleO OpenStack deployment, the undercloud need to control the overcloud power management, as well as serve its nodes with an operating system. Trying to do that inside an OpenStack cloud requires some modification from the client side as well as from the OpenStack cloud

The OVB (openstack virtual baremetal) project solves this problem and we strongly recommended to read its documentation prior to moving next in this document.

OVB architecture overview

An OVB setup requires additional node to be present: Baremetal Controller (BMC). This nodes captures all the IPMI requests dedicated to the OVB nodes and handles the machine power on/off operations, boot device change and other operations performed during the introspection phase.

Network architecture overview:



The BMC node should be connected to the management network. *infrared* brings up an IP address on own management interface for every Overcloud node. This allows *infrared* to handle IPMI commands coming from the undercloud. Those IPs are later used in the generated `instackenv.json` file.

For example, during the introspection phase, when the BMC sees the power off request for the OVB1 node, it performs a shutdown for the instance which corresponds to the OVB1 on the host cloud.

Provision ovb nodes

In order to provision ovb nodes, the openstack provisioner can be used:

```
ir openstack -vvvv -o provision.yml \
    --cloud=qeos7 \
    --prefix=example-ovb- \
    --topology-nodes=ovb_undercloud:1,bmc:1,ovb_controller:1,ovb_compute:1 \
    --topology-network=3_nets_ovb  \
    --key-file ~/.ssh/example-key.pem \
    --key-name=example-jenkins \
    --image=rhel-guest-image-7.3-35_3nics
```

The `--topology-nodes` options should include the `bmc` instance. Also instead of standard `compute` and `controller` nodes the appropriate nodes with the `ovb` prefix should be used. Such ovb node settings file holds several additional properties:

- instance `image` details. Currently the `ipxe-boot` image should be used for all the ovb nodes. Only that image allows to boot from the network after restart.
- `ovb` group in the groups section
- network topology (NICs' order)

For example, the ovb_compute settings can hold the following properties:

```
node_dict:
    name: compute
    image:
        name: "ipxe-boot"
        ssh_user: "root"
    interfaces:
        nic1:
```

```
            network: "data"
        nic2:
            network: "management"
        nic3:
            network: "external"
    external_network: external

    groups:
        - compute
        - openstack_nodes
        - overcloud_nodes
        - ovb
```

The `--topology-network` should specify the topology with at 3 networks: `data`, `management` and `external`:

- data network is used by the TripleO to provision the overcloud nodes
- management is used by the BMC to control IPMI operations
- external holds floating ip's and used by *infrared* to access the nodes

DHCP should be enabled only for the external network.

infrared provides the default `3_nets_ovb` network topology that allows to deploy the OVB setup.

The `--image` option should point to an image with 3 NICs configured on boot (as opposed the the default single NIC in common cloud images). The NICs' order should correspond to the network topology described in the ovb node settings files.

Install OpenStack with TripleO

To install OpenStack on ovb nodes the process is almost standard with small deviation.

The undercloud can be installed by running:

```
infrared tripleo-undercloud -v \
    --version 10 \
    --images-task rpm
```

The overcloud installation can be run with:

```
infrared tripleo-overcloud -v \
    --version 10 \
    --deployment-files ovb \
    --public-network=yes \
    --public-subnet=ovb_subnet \
    --network-protocol ipv4 \
    --post=yes \
    --introspect=yes \
    --tagging=yes
```

Here some ovb specific option should be considered:

- if host cloud is not patched and not configured for the OVB deployments the `--deployment-files` should point to the ovb templates to skip unsupported features. See the *OVB limitations* for details
- the `--public_subnet` should point to the subnet settings to match with the OVB network topology and allocation addresses

Fully functional overcloud will be deployed into the OVB nodes.

OVB limitations

The OVB approach requires a host cloud to be patched and configured. Otherwise the following features will **NOT** be available:

- Network isolation
- HA (high availability). Setup with more that 1 controller, etc is not allowed.
- Boot from network. This can be workaround by using the ipxe_boot image for the OVB nodes.

Troubleshoot

This page will list common pitfalls or known issues, and how to overcome them

Ansible Failures

Unreachable

Symptoms:

```
fatal: [hypervisor]: UNREACHABLE! => {"changed": false, "msg": "Failed to connect to␣
↪the host via ssh.", "unreachable": true}
```

Solution:

When Ansible fails because of `UNREACHABLE` reason, try to validate SSH credentials and make sure that all host are SSH-able.

In the case of `virsh` plugin, it's clear from the message above that the designated hypervisor is unreachable. Check that:

1. `--host-address` is a reachable address (IP or FQDN).
2. `--host-key` is a **private** (not **public**) key file and that its permissions are correct.
3. `--host-user` (defaults to `root`) exits on the host.
4. Try to manually ssh to the host using the given user private key:

   ```
   ssh -i $HOST_KEY $HOST_USER@$HOST_ADDRESS
   ```

Virsh Failures

Cannot create VM's

Symptoms:

Virsh cannot create a VM and displays the following message:

```
ERROR    Unable to add bridge management port XXX: Device or resource busy
Domain installation does not appear to have been successful.
Otherwise, you can restart your domain by running:
  virsh --connect qemu:///system start compute-0
otherwise, please restart your installation.
```

Solution:

This often can be caused by the misconfiguration of the hypervisor. Check that all the `ovs` bridges are properly configured on the hypervisor:

```
$ ovs-vsctl show

6765bb7e-8f22-4dbe-848f-eaff2e94ed96
Bridge brbm
    Port "vnet1"
        Interface "vnet1"
            error: "could not open network device vnet1 (No such device)"
    Port brbm
        Interface brbm
            type: internal
ovs_version: "2.6.1"
```

To fix the problem remove the broken bridge:

```
$ ovs-vsctl del-br brbm
```

Frequently Asked Questions

Where's my inventory file?

I'd like to run some personal Ansible playbook and/or "ad-hoc" but I can't find my inventory file

All Ansible environment files are read from, and written to, workspaces Use `infrared workspace inventory` to fetch a symlink to the active workspace's inventory or `infrared workspace inventory WORKSPACE` for any workspace by name:

```
ansible -i `infrared workspace inventory` all -m ping

compute-0 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
compute-1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
controller-0 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
```

```
}
hypervisor | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
undercloud-0 | SUCCESS => {
    "changed": false,
    "ping": "pong"
```

Beaker

Provision baremetal machines using Beaker.

Required arguments

- `--url`: URL of the Beaker server.
- `--password`: The password for the login user.
- `--host-address`: Address/FQDN of the baremetal machine to be provisioned.

Optional arguments

- `--user`: Login username to authenticate to Beaker. (default: admin)
- `--web-service`: For cases where the Beaker user is not part of the kerberos system, there is a need to set the Web service to RPC for authentication rather than rest. (default: rest)
- `--ca-cert`: For cases where the beaker user is not part of the kerberos system, a CA Certificate is required for authentication with the Beaker server.
- `--host-user`: The username to SSH to the host with. (default: root)
- `--host-password`: User's SSH password
- `--host-key`: User's SSH key
- `--image`: The image to use for nodes provisioning. (Check the "sample.yml.example" under vars/image for example)
- `--cleanup`: Release the system

Note: Please run `ir beaker --help` for a full detailed list of all available options.

Execution example

Provision:

```
ir beaker --url=beaker.server.url --user=beaker.user --password=beaker.password --
↪host-address=host.to.be.provisioned
```

Cleanup (Used for returning a loaned machine):

```
ir beaker --url=beaker.server.url --user=beaker.user --password=beaker.password --
↪host-address=host.to.be.provisioned --cleanup=yes
```

Foreman

Provision baremetal machine using Foreman and add it to the inventory file.

Required arguments

- `--url`: The Foreman API URL.
- `--user`: Foreman server login user.
- `--password`: Password for login user
- `--host-address`: Name or ID of the target host as listed in the Foreman server.

Optional arguments

- `--strategy`: Whether to use Foreman or system `ipmi` command. (default: foreman)
- `--action`: Which command to send with the power-management selected by mgmt_strategy. (default: cycle)
- `--wait`: Whether wait for host to return from rebuild or not. (default: yes)
- `--host-user`: The username to SSH to the host with. (default: root)
- `--host-password`: User's SSH password
- `--host-key`: User's SSH key
- `--host-ipmi-username`: Host IPMI username.
- `--host-ipmi-password`: Host IPMI password.
- `--roles`: Host roles

Note: Please run `ir foreman --help` for a full detailed list of all available options.

Execution example

```
ir foreman --url=foreman.server.api.url --user=foreman.user --password=foreman.
↪password --host-address=host.to.be.provisioned
```

OpenStack

Provision VMs on an exiting OpenStack cloud, using native ansible's cloud modules.

OpenStack Cloud Details

- **--cloud: reference to OpenStack cloud credentials, using os-client-config** This library expects properly configured `cloud.yml` file:

Listing 1.2: clouds.yml

```
clouds:
    cloud_name:
        auth_url: http://openstack_instance:5000/v2.0
        username: <username>
        password: <password>
        project_name: <project_name>
```

`cloud_name` can be then referenced with `--cloud` option:

```
infrared openstack --cloud cloud_name ...
```

`clouds.yml` is expected in either `~/.config/openstack` or `/etc/openstack` directories according to documentation:

Note: You can also omit the cloud parameter, and *infrared* will sourced openstackrc file:

```
source keystonerc
infrared openstack openstack ...
```

- **--key-file: Private key that will be used to ssh to the provisioned VMs.** Chosen matching public key will be uploaded to the OpenStack account, unless `--key-name` is provided
- **--key-name: Name of an existing *keypair* under the OpenStack account.** *keypair* should hold the public key that matches the provided private `--key-file`. Use `openstack --os-cloud cloud_name keypair list` to list available keypairs.
- **--dns: A Local DNS server used for the provisioned networks and VMs.** If not provided, OpenStack will use default DNS settings, which, in most cases, will not resolve internal URLs.

Topology

- **--prefix: prefix all resources with a string.** Use this with shared tenants to have unique resource names.

Note: `--prefix "XYZ"` will create router named `XYZrouter`. Use `--prefix "XYZ-"` to create `XYZ-router`

- **--topology-network: Description of the network topology.** By default, 3 networks will be provisioned with 1 router. 2 of them will be connected via the router to an external network discovered automatically (when more than 1 external network is found, the first will be chosen).
- **--topology-nodes: *KeyValueList* description of the nodes.** A floating IP will be provisioned on a designated network.

For more information about the structure of the topology files and how to create your own, please refer to Topology and Virsh plugin description.

- **--image: default image name or id for the VMs** use `openstack --os-cloud cloud_name image list` to see a list of available images

- **`--cleanup` Boolean. Whether to provision resources, or clean them from the tenant.** *Infrared* registers all provisioned resources to the workspace on creation, and will clean only registered resources:

```
infrared openstack --cleanup yes
```

VIRSH

Virsh provisioner is explicitly designed to be used for setup of virtual environments. Such environments are used to emulate production environment like tripleo-undercloud instances on one baremetal machine. It requires one prepared baremetal host (designated `hypervisor`) to be reachable through SSH initially.

Hypervisor machine

Hypervisor machine is the target machine where *infrared*'s virsh provisioner will create virtual machines and networks (using libvirt) to emulate baremetal infrastructure.

As such there are several specific requirements it has to meet.

Generally, It needs to have **enough memory and disk** storage to hold multiple decent VMs (each with GBytes of RAM and dozens of GB of disk). Also for acceptable responsiveness (speed of deployment/testing) just <4 threads or low GHz CPU is not a recommended choice (if you have old and weaker CPU than current mid-high end mobile phone CPU you may suffer performance wise - and so more timeouts during deployment or in tests).

Especially, for Ironic (TripleO) to control them, those **libvirt VMs** need to be bootable/controllable for **iPXE provisioning**. And also extra user has to exist, which can ssh in the hypervisor and control (restart...) libvirt VMs.

Note: *infrared* is attempting to configure or validate all (most) of this but it's scattered across all provisiner/installer steps. Due to nature of installers such as OSPd and current'infrared' structure it may not be 100% safe for rerunning (failure in previous run may prevent following one from succeeding in these preparation steps). We are working on a more idempotent approach which should resolve the above issues (if present).

What **user has to provide**:

- have machine with **sudoer user ssh access** and **enough resources**, as minimum requirements for one VM are:
  - VCPU: 2|4|8
  - RAM: 8|16
  - HDD: 40GB+
  - in practice disk may be smaller, as they are thin provisioned, as long as you don't force writing all the data (aka Tempest with rhel-guest instead of cirros etc)
- **RHEL-7.3** is tested, **CentOS** is also expected to work
  - may work with other distributions (best-effort/limited support)
- **yum repositories** has to be **preconfigured** by user (foreman/...) before using *infrared* so it can install dependencies
  - esp. for *infrared* to handle `ipxe-roms-qemu` it requires either **RHEL-7.3-server channel**

What **infrared takes care of**:

- `ipxe-roms-qemu` package of at least `version 2016xxyy` needs to be installed
- other basic packages installed

  - `libvirt`, `libguestfs{-tools,-xfs}`, `qemu-kvm`, `wget`, `virt-install`
- **virtualization support** (VT-x/AMD-V)
  - ideally with **nested=1** support
- `stack` user created with polkit privileges for *org.libvirt.unix.manage*
- **ssh key** with which *infrared* can authenticate (created and) added for *root* and *stack* user, ATM they are handled differently/separately:
  - for *root* the `infared/id_rsa.pub` gets added to authorized_keys
  - for *stack* `infrared/id_rsa_undercloud.pub` is added to authorized_keys, created/added later during installation

First, Libvirt and KVM are installed and configured to provide a virtualized environment. Then, virtual machines are created for all requested nodes.

Topology

The first thing you need to decide before you deploy your environment is the `Topology`. This refers to the number and type of VMs in your desired deployment environment. If we use OpenStack as an example, a topology may look something like:

- 1 VM called undercloud
- 1 VM called controller
- 1 VM called compute

To control how each VM is created, we have creates a YAML file that describes the specification of each VM. For more information about the structure of the topology files and how to create your own, please refer to Topology.

Please see Bootstrap guide where usage is demonstrated.

Network layout

Baremetal machine used as host for such setup is called *hypervisor*. The whole deployment is designed to work within boundaries of this machine and (except public/natted traffic) shouldn't reach beyond. The following layout is part of default setup defined in plugins defaults:

```
      hypervisor
          |
          +--------+ nic0 - public IP
          |
          +--------+ nic1 - not managed
          |
            ...                                                 Libvirt VM's
          |                                                           |
    ------+--------+ data bridge (ctlplane, 192.0.2/24)              +------+ data␣
↪(nic0)
    |     |                                                           |
libvirt --+--------+ management bridge (nat, dhcp, 172.16.0/24)      +------+␣
↪managementnt (nic1)
    |     |                                                           |
    ------+--------+ external bridge (nat, dhcp, 10.0.0/24)          +------+ external␣
↪(nic2)
```

On *hypervisor*, there are 3 new bridges created with libvirt - data, management and external. Most important is data network which does not have DHCP and NAT enabled. This network can later be used as `ctlplane` for OSP director deployments (tripleo-undercloud). Other (usually physical) interfaces are not used (nic0, nic1, ...) except for public/natted traffic. External network is used for SSH forwarding so client (or Ansible) can access dynamically created nodes.

NAT Forwarding

By default, all networks above are NATed, meaning that they private networks only reachable via the *hypervisor* node. *infrared* configures the nodes SSH connection to use the *hypervisor* host as proxy.

Bridged Network

Some use-cases call for direct access to some of the nodes. This is achieved by adding a network with `forward: bridge` in its attributes to the network-topology file, and marking this network as external network on the relevant node files.

The result will create a virtual bridge on the *hypervisor* connected to the main NIC. VMs attached to this bridge will be served by the same LAN as the *hypervisor*.

> **Warning:** Be careful when using this feature. For example, an `undercloud` connected in this manner can disrupt the LAN by serving as an unauthorized DHCP server.

Fore example, see `tripleo` node used in conjunction with `3_net_1_bridge` network file:

```
infrared virsh [...] --topology-nodes ironic:1,[...] --topology-network 3_net_1_
↪bridge [...]
```

Workflow

1. Setup libvirt and kvm environment
2. Setup libvirt networks
3. Download base image for undercloud (`--image-url`)
4. Create desired amount of images and integrate to libvirt
5. Define virtual machines with requested parameters (`--topology-nodes`)
6. Start virtual machines

Environments prepared such in way are usually used as basic virtual infrastructure for tripleo-undercloud.

Note: Virsh provisioner has idempotency issues, so `infrared virsh ... --cleanup` must be run before reprovisioning every time.

TripleO Undercloud

Deploys a TripleO undercloud

Setup an Undercloud

- **`--version`: TripleO release to install.** Accepts either an integer for RHEL-OSP release, or a community release name (`Liberty`, `Mitaka`, `Newton`, etc...) for RDO release
- **`--build`: Specify a build date or a label for the repositories.** Supports any rhos-release labels. Examples: `passed_phase1`, `2016-08-11.1`, `Y1`, `Z3`, `GA` Not used in case of RDO.
- `--buildmods`: Let you the option to add flags to rhos-release:

  `pin` - Pin puddle (dereference 'latest' links to prevent content from changing). This flad is selected by default

  `flea` - Enable flea repos.

  `unstable` - This will enable brew repos or poodles (in old releases).

  `none` - Use none of those flags.

Note: `--buildmods` and `--build` flags are internal Red Hat users only.

- **`--cdn` Register the undercloud with a Red Hat Subscription Management platform.** Accepts a file with subscription details.

  Listing 1.3: cdn_creds.yml

  ```
  server_hostname: example.redhat.com
  username: user
  password: HIDDEN_PASS
  autosubscribe: yes
  server_insecure: yes
  ```

  For the full list of supported input, see the module documentation.

  Note: Pre-registered undercloud are also supported if `--cdn` flag is missing.

  Warning: The contents of the file are hidden from the logged output, to protect private account credentials.

To deploy a working undercloud:

```
infrared tripleo-undercloud --version 10
```

For better fine-tuning of packages, see *custom repositories*.

Overcloud Images

The final part of the undercloud installation calls for creating the images from which the OverCloud will be later created.

- Depending on `--images-task` these the undercloud can be either:
  - **`build` images:** Build the overcloud images from a fresh guest image. To use a different image than the default CentOS cloud guest image, use `--images-url` to define base image than CentOS. For OSP installation, you must provide a url of a valid RHEL image.

  - **`import` images from url:** Download pre-built images from a given `--images-url`.
  - **Download images via `rpm`:** Starting from OSP 8, TripleO is packages with pre-built images avialable via RPM. .. note:: This option is invalid for *RDO* installation.

- Use `--images-repos` to instruct infrared whether to inject the repositories defined in the setup stage to the image (Allowing later update of the OverCloud)
- Use `--images-packages` to define a list of additional packages to install on the OverCloud image
- `--images-cleanup` tells *infrared* do remove the images files original after they are uploaded to the undercloud's Glance service.

To configure overcloud images:

```
infrared tripleo-undercloud --images-task rpm
```

Note: This assumes an undercloud was already installed and will skip installation stage because `--version` is missing.

When using RDO (or for OSP 7), `rpm` strategy in unavailable. Use `import` with `--images-url` to download overcloud images from web:

```
infrared tripleo-undercloud --images-task import --images-url http://buildlogs.centos.
↪org/centos/7/cloud/x86_64/tripleo_images/mitaka/delorean
```

Note: The RDO overcloud images can be also found here: https://images.rdoproject.org

If pre-packaged images are unavailable, tripleo can build the images locally on top of a regular cloud guest image:

```
infrared tripleo-undercloud --images-task build
```

CentOS or RHEL guest images will be used for RDO and OSP respectively. To use a different image specify `--images-url`:

```
infrared tripleo-undercloud --images-task build --images-url http://cloud.centos.org/
↪centos/7/images/CentOS-7-x86_64-GenericCloud.qcow2
```

Note: building the images takes a long time and it's usually quicker to download them.

See the RDO deployment page for more details on how to setup RDO product.

Undercloud Configuration

Undercloud is configured according to `undercloud.conf` file. Use `--config-file` to provide this file, or let *infrared* generate one automatically, based on a sample file provided by the project. Use `--config-options` to provide a list of `section.option=value` that will override specific fields in it.

Use the `--ssl=yes` option to install enable SSL on the undercloud. If used, a self-signed SSL cert will be generated.

Custom Repositories

Add custom repositories to the undercloud, after installing the TripleO repositories.

- **`--repos-config` setup repos using the ansible yum_repository module.** Using this option enables you to set specific options for each repository:

Listing 1.4: repos_config.yml

```
---
extra_repos:
    - name: my_repo1
      file: my_repo1.file
      description: my repo1
      base_url: http://myurl.com/my_repo1
      enabled: 0
      gpg_check: 0
    - name: my_repo2
      file: my_repo2.file
      description: my repo2
      base_url: http://myurl.com/my_repo2
      enabled: 0
      gpg_check: 0
    ...
```

Note: This expicitly supports some of the options found in yum_repository module (name, file, description, base_url, enabled and gpg_check). For more information about this module, visit Ansible yum_repository documentation.

- `repos-urls`: comma separated list of URLs to download repo files to `/etc/yum.repos.d`

Both options can be used togather:

```
infrared tripleo-undercloud [...] --repos-config repos_config.yml --repos-urls "http:/
↪/yourul.com/repofile1.repo,http://yourul.com/repofile2.repo"
```

TripleO Undercloud User

`--user-name` and `--user-password` define a user, with password, for the undercloud. Acorrding to TripleO guidelines, the default username is `stack`. User will be created if necessary.

Backup

When working on a virtual environment, *infrared* can create a snapshot of the installed undercloud that can be later used to *restore* it on a future run, thus saving installation time.

In order to use this feature, first follow the *Setup an Undercloud* section. Once an undercloud VM is up and ready, run the following:

```
ir tripleo-undercloud --quickstart-backup yes
```

Or optionally, provide the file name of the image to create (defaults to "undercloud-quickstart.qcow2"). .. note:: the filename refers to a path on the hypervisor.

ir tripleo-undercloud –quickstart-backup yes –quickstart-filename custom-name.qcow2

This will prepare a qcow2 image of your undercloud ready for usage with *Restore*.

Note: this assumes an undercloud is already installed and will skip installation and images stages.

Restore

When working on a virtual environment, *infrared* can use a pre-made undercloud image to quickly set up an environment. To use this feature, simply run:

```
ir tripleo-undercloud --quickstart-restore yes
```

Or optionally, provide the file name of the image to restore from (defaults to "undercloud-quickstart.qcow2"). .. note:: the filename refers to a path on the hypervisor.

Undercloud Upgrade

Upgrade is discovering current Undercloud version and upgrade it to the next major one. To upgrade Undercloud run the following command:

```
infrared tripleo-undercloud -v --upgrade yes
```

Note: The Overcloud won't need new images to upgrade to. But you'd need to upgrade the images for OC nodes before you attempt to scale out nodes. Example for Undercloud upgrade and images update:

```
infrared tripleo-undercloud -v --upgrade yes --images-task rpm
```

Warning: Currently, there is upgrade possibility from version 9 to version 10 only.

Undercloud Update

Update is discovering current Undercloud version and perform minor version update. To update Undercloud run the following command:

```
infrared tripleo-undercloud -v --update-undercloud yes
```

Example for update of Undercloud and Images:

```
infrared tripleo-undercloud -v --update-undercloud yes --images-task rpm
```

Warning: Infrared support update for RHOSP from version 8.

TripleO Overcloud

Deploys a TripleO overcloud from an existing undercloud

Stages Control

Run is broken into the following stages. Omitting any of the flags (or setting it to `no`) will skip that stage

- `--introspect` the overcloud nodes
- `--tag` overcloud nodes with proper flavors
- `--deploy` overcloud of given `--version` (see below)
- Execute `--post` installation steps (like creating a public network - see below)

Deployment Description

- **`--deployment-files`: Mandatory.** Path to a directory, containing heat-templates describing the overcloud deployment. Choose `virt` to enable preset templates for virtual POC environment (virsh or ovb).
- **`--instackenv-file`:** Path to the instackenv.json configuration file used for introspection. For virsh and ovb deployment, *infrared* can generate this file automatically.
- **`--version`: TripleO release to install.** Accepts either an integer for RHEL-OSP release, or a community release name (`Liberty`, `Mitaka`, `Newton`, etc...) for RDO release
- **The following options define the number of nodes in the overcloud:** `--controller-nodes`, `--compute-nodes`, `--storage-nodes`. If not provided, will try to evaluate the exiting nodes and default to `1` for `compute`/`controller` or `0` for `storage`.

Overcloud Options

- `--overcloud-ssl`: Boolean. Enable SSL for the overcloud services.
- `--overcloud-debug`: Boolean. Enable debug mode for the overcloud services.
- **`--overcloud-templates`: Add extra environment template files or custom templates** to "overcloud deploy" command. Format:

Listing 1.5: sahara.yml

```
---
tripleo_heat_templates:
    - /usr/share/openstack-tripleo-heat-templates/environments/services/
↪sahara.yaml
```

Listing 1.6: ovs-security-groups.yml

```
---
tripleo_heat_templates:
    []

custom_templates:
    parameter_defaults:
        NeutronOVSFirewallDriver: openvswitch
```

- **`--overcloud-script`: Customize the script that will deploy the overcloud.** A path to a `*.sh` file containing `openstack overcloud deploy` command. This is for advance users.
- **`--heat-templates-basedir`: Allows to override the templates base dir** to be used for deployment. Default value: "/usr/share/openstack-tripleo-heat-templates"

Overcloud Public Network

- `--public-network`: Bool. Whether to have *infrared* create a public network on the overcloud.
- **`--public-subnet:`** Path to file containing different values for the subnet of the network above.
- **`--public-vlan:`** Set this to `yes` if overcloud's external network is on a VLAN that's unreachable from the undercloud. This will configure network access from UnderCloud to overcloud's API/External(floating IPs) network, creating a new VLAN interface connected to ovs's `br-ctlplane` bridge. !NOTE: If your UnderCloud's network is already configured properly, this could disrupt it, making overcloud API unreachable For more details, see: VALIDATING THE OVERCLOUD

Overcloud Storage

- **`--storage-external:` Bool** If `no`, the overcloud will deploy and manage the storage nodes. If `yes` the overcloud will connect to an external, per-existing storage service.
- **`--storage-backend:`** The type of storage service used as backend.
- **`--storage-config:`** Storage configuration (YAML) file.

Composable Roles

InfraRed allows to use custom roles to deploy overcloud. Check the Composable roles page for details.

Overcloud Upgrade

Warning: Before Overcloud upgrade you need to perform upgrade of Undercloud

Upgrade will detect Undercloud version and will upgrade Overcloud to the same version.

- `--upgrade`: Bool If *yes*, the overcloud will be upgraded.

Example:

```
infrared tripleo-overcloud -v --upgrade yes --deployment-files virt
```

- `--updateto`: target build to upgrade to

Example:

```
infrared tripleo-overcloud -v --upgrade yes --updateto 2017-05-30.1 --deployment-
↪files virt
```

Note: Upgrade is assuming that Overcloud Deployment script and files/templates, which were used during the initial deployment are available at Undercloud node in home directory of Undercloud user. Deployment script location is assumed to be "~/overcloud_deploy.sh"

Overcloud Update

Warning: Before Overcloud update it's recommended to update Undercloud

Note: InfraRed supports minor updates from OpenStack 9

Minor update detects Undercloud's version and updates packages within same version to latest available.

- `--updateto`: target build to update to defaults to `None`, in which case, update is disabled. possible values: build-date, `latest`, `passed_phase1`, `z3` and all other labels supported by `rhos-release` When specified, rhos-release repos would be setup and used for minor updates.

Example:

```
infrared tripleo-overcloud -v --updateto latest --deployment-files virt
```

Note: Minor update expects that Overcloud Deployment script and files/templates, used during the initial deployment, are available at Undercloud node in home directory of Undercloud user. Deployment script location is assumed to be "~/overcloud_deploy.sh"

- `--buildmods`: Let you the option to add flags to rhos-release:

  `pin` - Pin puddle (dereference 'latest' links to prevent content from changing). This flag is selected by default

  `flea` - Enable flea repos.

  `unstable` - This will enable brew repos or poodles (in old releases).

  `none` - Use none of those flags.

  Note: `--buildmods` flag is for internal Red Hat usage.

Tempest

Runs Tempest tests against an OpenStack cloud.

Required arguments

- **`--openstack-installer`: The installer used to deploy OpenStack.** Enables extra configuration steps for certain installers. Supported installers are: `tripleo` and `packstack`.
- **`--openstack-version`: The version of the OpenStack installed.** Enables additional configuration steps when version <= 7.
- **`--tests`: The list of test suites to execute. For example: `network,compute`.** The complete list of the available suites can be found by running `ir tempest --help`
- **`--openstackrc`: The OpenStack RC file.** The absolute and relative paths to the file are supported. When this option is not provided, *infrared* will try to use the *keystonerc* file from the active workspace. The openstackrc file is copied to the tester station and used to configure and run Tempest.

Optional arguments

The following useful arguments can be provided to tune tempest tester. Complete list of arguments can be found by running `ir tempest --help`.

- **`--setup`: The setup type for the tempest.** Can be `git` (default), `rpm` or pip. Default tempest git repository is https://git.openstack.org/openstack/tempest.git. This value can be overridden with the `--extra-vars` cli option:

```
ir tempest -e setup.repo=my.custom.repo [...]
```

- **`--revision`: Specifies the revision for the case when tempest is installing from the git repository.** Default value is `HEAD`.
- **`--deployer-input-file`: The deployer input file to use for Tempest configuration.** The absolute and relative paths to the file are supported. When this option is not provided *infrared* will try to use the *deployer-input-file.conf* file from active workspace folder.

  For some OpenStack versions(kilo, juno, liberty) Tempest provides predefined deployer files. Those files can be downloaded from the git repo and passed to the Tempest tester:

```
BRANCH=liberty
wget https://raw.githubusercontent.com/redhat-openstack/tempest/$BRANCH/etc/
↪deployer-input-$BRANCH.conf
ir tempest --tests=sanity \
           --openstack-version=8 \
           --openstack-installer=tripleo \
           --deployer-input-file=deployer-input-$BRANCH.conf
```

- **`--image`: Image to be uploaded to glance and used for testing. Path have to be a url.** If image is not provided, tempest config will use the default.

Note: You can specify image ssh user with `--config-options compute.image_ssh_user=`

Tempest results

infrared fetches all the tempest output files, like results to the `tempest_results` folder under the active workspace folder:

```
ll .workspace/my_workspace/tempest_results/tempest-*

-rw-rw-r--. tempest-results-minimal.xml
-rw-rw-r--. tempest-results-neutron.xml
```

Collect-logs

Collect-logs plugin allows the user to collect files & directories from hosts managed by active workspace. A list of paths to be archived is taken from `vars/default_archives_list.yml` in the plugin's dir. Logs are being packed as `.tar` files by default, unless the user explicitly use the `--gzip` flag that will instruct the plugin to compress the logs with `gzip`.

Note: All nodes must have yum repositories configured in order for the tasks to work on them.

Note: Users can manually edit the `default_archives_list.yml` if need to add/delete paths.

Usage example:

```
ir collect-logs --dest-dir=/tmp/ir_logs
```

Gabbi Tester

Runs telemetry tests against the OpenStack cloud.

Required arguments

- **`--openstack-version`: The version of the OpenStack installed.** That option also defines the list of tests to run against the OpenStack.
- **`--openstackrc`: The OpenStack RC file.** The absolute and relative paths to the file are supported. When this option is not provided, *infrared* will try to use the *keystonerc* file from the active workspace. The openstackrc file is copied to the tester station and used to run tests
- **`--undercloudrc`: The undercloud RC file.** The absolute and relative paths to the file are supported. When this option is not provided, *infrared* will try to use the *stackrc* file from the active workspace.

Optional arguments

- `--network`: Network settings to use. Default network configuration includes the `protocol` (ipv4 or ipv6) and `interfaces` sections:

```
network:
    protocol: ipv4
    interfaces:
        - net: management
          name: eth1
        - net: external
          name: eth2
```

- `--setup`: The setup variables, such as git repo name, folders to use on tester and others:

```
setup:
    repo_dest: ~/TelemetryGabbits
    gabbi_venv: ~/gbr
    gabbits_repo: <private-repo-url>
```

Gabbi results

infrared fetches all the output files, like results to the `gabbi_results` folder under the active workspace folder.

RDO deployment

Infrared allows to perform RDO based deployments.

To deploy RDO on virtual environment the following steps can be performed.

1. Provision virtual machines on a hypervisor with the virsh plugin. Use CentOS image:

```
infrared virsh -vv \
    -o provision.yml \
    --topology-nodes undercloud:1,controller:1,compute:1,ceph:1  \
    --host-address my.host.redhat.com \
    --host-key /path/to/host/key \
    --image-url https://cloud.centos.org/centos/7/images/CentOS-7-x86_64-
↪GenericCloud.qcow2 \
    -e override.controller.cpu=8 \
    -e override.controller.memory=32768
```

2. Install the undercloud. Use RDO release name as a version:

```
infrared tripleo-undercloud -vv -o install.yml \
    -o undercloud-install.yml \
    --version ocata
```

3. Build or import overcloud images from https://images.rdoproject.org:

```
# import images
infrared tripleo-undercloud -vv \
    -o undercloud-images.yml \
    --images-task=import \
    --images-url=https://images.rdoproject.org/ocata/rdo_trunk/current-tripleo/
↪stable/

# or build images
infrared tripleo-undercloud -vv \
    -o undercloud-images.yml \
    --images-task=build \
```

Note: Overcloud image build process often takes more time than import.

4. Install RDO:

```
infrared tripleo-overcloud -v \
    -o overcloud-install.yml \
    --version ocata \
    --deployment-files virt \
    --introspect yes \
    --tagging yes \
    --deploy yes \
    --post yes
```

Known issues

1. Overcloud deployment fails with the following message:

```
Error: /Stage[main]/Gnocchi::Db::Sync/Exec[gnocchi-db-sync]: Failed to call␣
↪refresh: Command exceeded timeout
Error: /Stage[main]/Gnocchi::Db::Sync/Exec[gnocchi-db-sync]: Command exceeded␣
↪timeout
```

This error might be caused by the https://bugs.launchpad.net/tripleo/+bug/1695760. To workaround that issue the `--overcloud-templates disable-telemetry` flag should be added to the tripleo-overcloud command:

```
infrared tripleo-overcloud -v \
    -o overcloud-install.yml \
    --version ocata \
    --deployment-files virt \
    --introspect yes \
    --tagging yes \
    --deploy yes \
    --post yes \
    --overcloud-templates disable-telemetry
```

Composable Roles

InfraRed allows to define Composable Roles while installing OpenStack with tripleo.

Overview

To deploy overcloud with the composable roles the additional templates should be provided:

- nodes template: list all the roles, list of the services for every role. For example:

```
- name: ObjectStorage
  CountDefault: 1
  ServicesDefault:
      - OS::TripleO::Services::CACerts
      - OS::TripleO::Services::Kernel
      - OS::TripleO::Services::Ntp
      [...]
  HostnameFormatDefault: swift-%index%

- name: Controller
  CountDefault: 1
  ServicesDefault:
      - OS::TripleO::Services::CACerts
      - OS::TripleO::Services::CephMon
      - OS::TripleO::Services::CephExternal
      - OS::TripleO::Services::CephRgw
      [...]
  HostnameFormatDefault: controller-%index%

- name: Compute
  CountDefault: 1
  ServicesDefault:
      - OS::TripleO::Services::CACerts
      - OS::TripleO::Services::CephClient
      - OS::TripleO::Services::CephExternal
```

```
      [....]
  HostnameFormatDefault: compute-%index%

- name: Networker
  CountDefault: 1
  ServicesDefault:
      - OS::TripleO::Services::CACerts
      - OS::TripleO::Services::Kernel
      [...]
  HostnameFormatDefault: networker-%index%
```

- template with the information about roles count, flavors and other defaults:

```
parameter_defaults:
    ObjectStorageCount: 1
    OvercloudSwiftStorageFlavor: swift
    ControllerCount: 2
    OvercloudControlFlavor: controller
    ComputeCount: 1
    OvercloudComputeFlavor: compute
    NetworkerCount: 1
    OvercloudNetworkerFlavor: networker
    [...]
```

- template with the information about roles resources (usually network and port resources):

```
resource_registry:
    OS::TripleO::ObjectStorage::Net::SoftwareConfig: /home/stack/deployment_
↪files/network/nic-configs/swift-storage.yaml
    OS::TripleO::Controller::Net::SoftwareConfig: /home/stack/deployment_
↪files/network/nic-configs/controller.yaml
    OS::TripleO::Compute::Net::SoftwareConfig: /home/stack/deployment_files/
↪network/nic-configs/compute.yaml
    OS::TripleO::Networker::Ports::TenantPort: /usr/share/openstack-tripleo-
↪heat-templates/network/ports/tenant.yaml
    OS::TripleO::Networker::Ports::InternalApiPort: /usr/share/openstack-
↪tripleo-heat-templates/network/ports/internal_api.yaml
    OS::TripleO::Networker::Net::SoftwareConfig: /home/stack/deployment_files/
↪network/nic-configs/networker.yaml
    [...]
```

InfraRed allows to simplify the process of templates generation and auto-populates the roles according to the deployed topology.

Defining topology and roles

To deploy custom roles, InfraRed should know what nodes should be used for what roles. This involves a 2-step procedure.

Step #1 Setup available nodes and store them in the InfraRed inventory. Those nodes can be configured by the `provision` plugin such as virsh:

```
ir virsh -vvvv
    --topology-nodes=undercloud:1,controller:2,compute:1,networker:1,swift:1 \
    --host-address=seal52.qa.lab.tlv.redhat.com \
    --host-key ~/.ssh/my-prov-key
```

In that example we defined a `networker` nodes which holds all the neutron services.

Step #2 Provides a path to the roles definition while installing the overcloud using the `--role-files` option:

```
ir tripleo-overcloud -vvvv
    --version=10 \
    --deploy=yes \
    --role-files=networker \
    --deployment-files=composable_roles \
    --introspect=yes \
    --storage-backend=swift \
    --tagging=yes \
    --post=yes
```

In that example, to build the composable roles templates, InfraRed will look into the `<plugin_dir>/files/roles/networker` folder for the files that corresponds to all the node names defined in the `inventory->overcloud_nodes` group.

All those role files hold role parameters. See *Role Description* section for details.

When role file is not found in the user specified folder InfraRed will try to use a `default` roles from the `<plugin_dir>/files/roles/default` folder.

For the topology described above with the networker custom role the following role files can be defined:

- <plugin_dir>/files/roles/**networker**/controller.yml - holds controller roles without neutron services
- <plugin_dir>/files/roles/**networker**/networker.yml - holds the networker role description with the neutron services
- <plugin_dir>/files/roles/**default**/compute.yml - a default compute role description
- <plugin_dir>/files/roles/**default**/swift.yml - a default swift role description

To deploy non-supported roles, a new folder should be created in the `<plugin_dir>/files/roles/`. Any roles files that differ (e.g. service list) from the defaults should be put there. That folder is then can be referenced with the `--role-files=<folder name>` argument.

Role Description

All the custom and defaults role descriptions are stored in the `<plugin_dir>/files/roles` folder. Every role file holds the following information:

- `name` - name of the role
- `resource_registry` - all the resources required for a role.
- `flavor` - the flavor to use for a role
- `host_name_format` - the resulting host name format for the role node
- `services` - the list of services the role holds

Below is an example of the controller default role:

```
controller_role:
    name: Controller

    # the primary role will be listed first in the roles_data.yaml template file.
    primary_role: yes

    # include resources
```

```
    # the following vars can be used here:
    #  - ${ipv6_postfix}: will be replaced with _v6 when the ipv6 protocol is used␣
↪for installation, otherwise is empty
    #  - ${deployment_dir} - will be replaced by the deployment folder location on␣
↪the undercloud. Deployment folder can be specified with the ospd --deployment flag
    resource_registry:
        "OS::TripleO::Controller::Net::SoftwareConfig": "${deployment_dir}/network/
↪nic-configs/controller${ipv6_postfix}.yaml"

    # we can also set a specific flavor for a role.
    flavor: controller
    host_name_format: 'controller-%index%'

    # condition can be used to include or disable services. For example:
    #  - "{% if install.version |openstack_release < 11 %}
↪OS::TripleO::Services::VipHosts{% endif %}"
    services:
        - OS::TripleO::Services::CACerts
        - OS::TripleO::Services::CephClient
        - OS::TripleO::Services::CephExternal
        - OS::TripleO::Services::CephRgw
        - OS::TripleO::Services::CinderApi
        - OS::TripleO::Services::CinderBackup
        - OS::TripleO::Services::CinderScheduler
        - OS::TripleO::Services::CinderVolume
        - OS::TripleO::Services::Core
        - OS::TripleO::Services::Kernel
        - OS::TripleO::Services::Keystone
        - OS::TripleO::Services::GlanceApi
        - OS::TripleO::Services::GlanceRegistry
        - OS::TripleO::Services::HeatApi
        - OS::TripleO::Services::HeatApiCfn
        - OS::TripleO::Services::HeatApiCloudwatch
        - OS::TripleO::Services::HeatEngine
        - OS::TripleO::Services::MySQL
        - OS::TripleO::Services::NeutronDhcpAgent
        - OS::TripleO::Services::NeutronL3Agent
        - OS::TripleO::Services::NeutronMetadataAgent
        - OS::TripleO::Services::NeutronApi
        - OS::TripleO::Services::NeutronCorePlugin
        - OS::TripleO::Services::NeutronOvsAgent
        - OS::TripleO::Services::RabbitMQ
        - OS::TripleO::Services::HAproxy
        - OS::TripleO::Services::Keepalived
        - OS::TripleO::Services::Memcached
        - OS::TripleO::Services::Pacemaker
        - OS::TripleO::Services::Redis
        - OS::TripleO::Services::NovaConductor
        - OS::TripleO::Services::MongoDb
        - OS::TripleO::Services::NovaApi
        - OS::TripleO::Services::NovaMetadata
        - OS::TripleO::Services::NovaScheduler
        - OS::TripleO::Services::NovaConsoleauth
        - OS::TripleO::Services::NovaVncProxy
        - OS::TripleO::Services::Ntp
        - OS::TripleO::Services::SwiftProxy
        - OS::TripleO::Services::SwiftStorage
        - OS::TripleO::Services::SwiftRingBuilder
```

```
    - OS::TripleO::Services::Snmp
    - OS::TripleO::Services::Timezone
    - OS::TripleO::Services::CeilometerApi
    - OS::TripleO::Services::CeilometerCollector
    - OS::TripleO::Services::CeilometerExpirer
    - OS::TripleO::Services::CeilometerAgentCentral
    - OS::TripleO::Services::CeilometerAgentNotification
    - OS::TripleO::Services::Horizon
    - OS::TripleO::Services::GnocchiApi
    - OS::TripleO::Services::GnocchiMetricd
    - OS::TripleO::Services::GnocchiStatsd
    - OS::TripleO::Services::ManilaApi
    - OS::TripleO::Services::ManilaScheduler
    - OS::TripleO::Services::ManilaBackendGeneric
    - OS::TripleO::Services::ManilaBackendNetapp
    - OS::TripleO::Services::ManilaBackendCephFs
    - OS::TripleO::Services::ManilaShare
    - OS::TripleO::Services::AodhApi
    - OS::TripleO::Services::AodhEvaluator
    - OS::TripleO::Services::AodhNotifier
    - OS::TripleO::Services::AodhListener
    - OS::TripleO::Services::SaharaApi
    - OS::TripleO::Services::SaharaEngine
    - OS::TripleO::Services::IronicApi
    - OS::TripleO::Services::IronicConductor
    - OS::TripleO::Services::NovaIronic
    - OS::TripleO::Services::TripleoPackages
    - OS::TripleO::Services::TripleoFirewall
    - OS::TripleO::Services::OpenDaylightApi
    - OS::TripleO::Services::OpenDaylightOvs
    - OS::TripleO::Services::SensuClient
    - OS::TripleO::Services::FluentdClient
    - OS::TripleO::Services::VipHosts
```

The name of the role files should correspond to the node inventory name without prefix and index. For example, for `user-prefix-controller-0` the name of the role should be `controller.yml`.

Deployment example

To deploy OpenStack with composable roles on virtual environment the following steps can be performed.

1. Provision all the required virtual machines on a hypervizor with the virsh plugin:

```
infrared virsh -vv \
    -o provision.yml \
    --topology-nodes undercloud:1,controller:3,db:3,messaging:3,networker:2,
↪compute:1,ceph:1  \
    --host-address my.host.redhat.com \
    --host-key /path/to/host/key \
    -e override.controller.cpu=8 \
    -e override.controller.memory=32768
```

2. Install undercloud and overcloud images:

```
infrared tripleo-undercloud -vv -o install.yml \
    -o undercloud-install.yml \
```

```
    --version 11 \
    --images-task rpm
```

3. Install overcloud:

```
infrared tripleo-overcloud -vv \
    -o overcloud-install.yml \
    --version 11 \
    --role-files=composition \
    --deployment-files composable_roles \
    --introspect yes \
    --tagging yes \
    --deploy yes \
    --post yes
```

Tripleo OSP with Red Hat Subscriptions

Undercloud

To deploy OSP, the Undercloud must be registered to Red Hat channels. Define the subscription details:

Listing 1.7: undercloud_cdn.yml

```
---
server_hostname: 'subscription.rhsm.redhat.com'
username: 'infrared.user@example.com'
password: '123456'
autosubscribe: yes
server_insecure: yes
```

Warning: During run time, contents of the file are hidden from the logged output, to protect private account credentials.

For the full list of supported input, see the Ansible module documentation. For example, `autosubscribe: yes` can be replaced with `pool_id` or `pool: REGEX`, where `REGEX` is a regular expression that searches for matching available pools.

Note: Pre-registered undercloud is also supported if `--cdn` flag is missing.

Deploy your undercloud. It's recommended to use `--images-task rpm` to fetch pre-packaged images that are only available via Red Hat channels:

```
infrared tripleo-undercloud --version 11 --cdn undercloud_cdn.yml --images-task rpm
```

Warning: `--images-update` is not supported with cdn.

Overcloud

Once the undercloud is registered, the overcloud can be deployed. However, the overcloud nodes will not be registered and cannot receive updates. While the nodes can be later registered manually, Tripleo provides a way to register them automatically on deployment.

According to the guide there are 2 heat-templates required. They can be included, and their defaults overridden, using a custom templates file.

Listing 1.8: overcloud_cdn.yml

```
---
tripleo_heat_templates:
    - /usr/share/openstack-tripleo-heat-templates/extraconfig/pre_deploy/rhel-
↪registration/rhel-registration-resource-registry.yaml
    - /usr/share/openstack-tripleo-heat-templates/extraconfig/pre_deploy/rhel-
↪registration/environment-rhel-registration.yaml

custom_templates:
    parameter_defaults:
        rhel_reg_activation_key: ""
        rhel_reg_org: ""
        rhel_reg_pool_id: ""
        rhel_reg_method: "portal"
        rhel_reg_sat_url: ""
        rhel_reg_sat_repo: "rhel-7-server-rpms rhel-7-server-extras-rpms rhel-7-
↪server-rh-common-rpms rhel-ha-for-rhel-7-server-rpms rhel-7-server-openstack-10-rpms
↪"
        rhel_reg_repos: ""
        rhel_reg_auto_attach: ""
        rhel_reg_base_url: "https://cdn.redhat.com"
        rhel_reg_environment: ""
        rhel_reg_force: "true"
        rhel_reg_machine_name: ""
        rhel_reg_password: "123456"
        rhel_reg_release: ""
        rhel_reg_server_url: "subscription.rhsm.redhat.com"
        rhel_reg_service_level: ""
        rhel_reg_user: "infrared.user@example.com"
        rhel_reg_type: ""
        rhel_reg_http_proxy_host: ""
        rhel_reg_http_proxy_port: ""
        rhel_reg_http_proxy_username: ""
        rhel_reg_http_proxy_password: ""
```

Note: Please notice that the repos in the file above are for OSP 10

Deploy the overcloud with the custom templates file:

```
infrared tripleo-overcloud --version=11 --deployment-files=virt --introspect=yes --
↪tagging=yes  --deploy=yes --overcloud-templates overcloud_cdn.yml --post=yes
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search